4 EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

The subsequent text serves to bridge theory with practice, showcasing the tangible impacts and benefits of using CROW, MEWE, and WASM-MUTATE. In this chapter, we present two selected use cases that exploit Software Diversification through our technical contributions presented in Chapter 3. The selected cases are representative of applications of Software Diversification for WebAssembly in browsers and standalone engines. Besides, we select the cases based on their impact and novelty.

4.1 Offensive Diversification: Malware evasion

The primary black-hat usage of WebAssembly is cryptojacking [**?**]. Such WebAssembly code mines cryptocurrencies on users' browsers for the benefit of malicious actors and without the consent of the users [**?**]. The main reason for this phenomenon is that the core foundation of cryptojacking is: the faster, the better. In this context, WebAssembly, a binary instruction format designed to be portable and fast, is a feasible technology for implementing and distributing cryptojacking over the web. A Kaspersky report about the state of cryptojacking in the first three quarters of 2022 confirms the steady growth in the usage of cryptominers [**?**]. The report shows that Monero [**?**] is the most used cryptocurrency for cryptomining in the browser. Attackers might hide WebAssembly cryptominers [**?**] in multiple locations inside web applications.

Antivirus and browsers provide support for detecting cryptojacking. For example, the Firefox browser supports the detection of cryptomining by using deny lists [**?**]. The academic community also provides related work on detecting

[0] Comp. time 2023/10/05 09:57:06

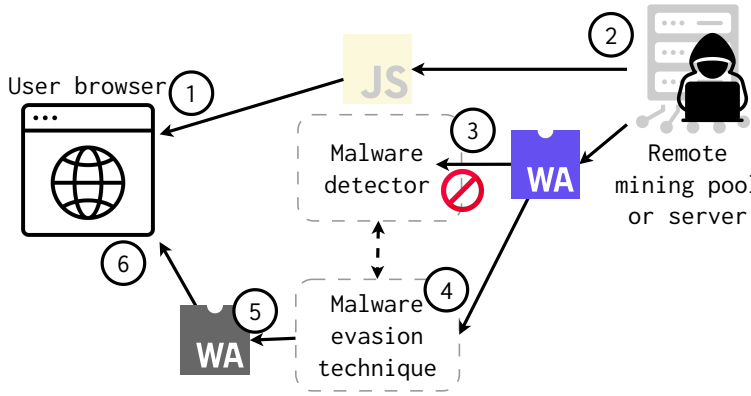

Figure 4.1: *Taken from [?]*

or preventing WebAssembly cryptojacking [? ? ? ? ? ?]. Yet, it is known that black-hats can use evasion techniques to bypass detection. Only the previous work of Bhansali et al. [?] investigates the possibility of WebAssembly cryptojacking to evade detection techniques. This is a crucial motivation for our work, one of the first to study WebAssembly malware evasion.

TODO Present the SOTA of malware evasion. The assumption on lack of diversity in the malware detectors, not that much, check authors. **TODO** Then present the threat model image here

4.1.1 Threat model and objective

In this paper, we design, implement and evaluate a full-fledged evasion pipeline for WebAssembly. Concretely, we use wasm-mutate as a diversifier [?], which implements 135 possible bytecode transformations, grouped into three categories: peephole operator, module structure, and control flow. We demonstrate the effectiveness of our evasion technique against two cryptojacking detectors: VirusTotal, a general detection tool that comprises 60 antiviruses and, MINOS [?], a WebAssembly-specific detector.

TODO Detail the threat model and make a clear statement of the objective of this use case.

Test and evade the resilience of WebAssembly malware detectors mentioned in Subsection 2.1.5.

4.1.2 Approach

We evaluate our proposed evasion technique on 33 cryptojacking malware that we curated from the 8643 binaries of the wasmbench dataset [?], to our

knowledge, the most exhaustive collection of real-world WebAssembly binaries. We experiment with the 33 binaries marked as potentially dangerous by at least one antivirus vendor of VirusTotal. We empirically demonstrate that evasion is possible for all of these 33 real-world WebAssembly cryptojacking malware while using a WebAssembly-specific detector. Remarkably, we find 30 cryptominers for which our technique successfully generates variants that evade VirusTotal. Our set of malware includes 6 cryptojacking programs that are fully reproducible in a controlled environment. With them, we assess that our evasion method does not affect malware correctness and generates fully functional malware variants with minimal overhead.

Our work provides evidence that the malware detection community has opportunities to strengthen the automatic detection of cryptojacking WebAssembly malware. The results of this work are actionable, as we provide quantitative evidence on specific malware transformations on which detection methods can focus.

TODO Mention the uncontrolled and controlled diversification. What are the difference between them. Make a statement about that obfuscation is a face of diversificaation. Specifically, controlled diversification.

TODO We use wasm-mutate **TODO** How do we use it? **TODO** Controlled and uncontrolled diversification.

4.1.3 Results

Contribution paper

The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WebAssembly Diversification for Malware Evasion" *at Computers & Security, 2023* `https://www.sciencedirect.com/science/article/pii/S0167404823002067`.

Discussion In this section we mention some challenges we face during the writing of this work. We enumerate them in order to enforce the debase and the discussion on the topic.

Oracle classification moves in time: One could expect that the more detectors a binary has, the more iterations are needed to evade them. However, we have observed that this is not the case for some binaries. The main reason is that the final labelling of binaries for VirusTotal vendors is not immediate [**?**]. For example, a VirusTotal vendor could label a binary as benign and change it later to malign after several weeks in a conservative way of acting. This phenomenon creates a time window in which slightly changed binaries (fewer iterations in our case) could evade the detection of numerous vendors.

Lack of bigger picture: A WebAssembly cryptomalware can only exist with its JavaScript complement. For example, a browser cryptonalware needs to send

the calculated hashes to a cryptocurrency service. This network communication is outside the WebAssembly accesses and needs to be delegated to a JavaScript code. Besides, other functionalities can be intermixing between JavaScript and WebAssembly and in some cases be completely in one side or the other [**?**]. This intermixing between JavaScript and WebAssembly could provide statically different WebAssembly. We have observed that, the imports and the memory data of the WebAssembly binaries have a high variability in our original dataset. The imported functions from JavaScript change from binary to binary. Their data segments can also differ in content and length. To completely analyze these cases, the whole JavaScript-WebAssembly program is needed, something only provided in 9/33 cases of our dataset.

More narrowed fitness function: We use a simple fitness function, but the MCMC evasion algorithm could have a fitness function as general as wanted. In our case, we do not use binary metadata, instead we focus on the result from the malware oracle, given that the main goal is to evade this oracle.

Mitigation: **TODO** TBD, data augmentation, better resilience evaluation ?

Another interesting thing would be to see if there is difference in the detectors. If some detectors are fooled by some transformations or are more robust, etc.

TODO Motivate the use case with the following sota

Binary rewriting tools and obfuscators The landscape for tools that can modify, obfuscate, or enhance WebAssembly binaries for various has increased. For instance, BREWasm[**?**] provides a comprehensive static binary rewriting framework specifically designed for WebAssembly. Wobfuscator[**?**] takes a different approach, serving as an opportunistic obfuscator for Wasm-JS browser applications. Madvex[**?**] focuses on modifying WebAssembly binaries to evade malware detection, with its approach being limited to alterations in the code section of a WebAssembly binary. Additionally, WASMixer[**?**] obfuscates WebAssembly binaries, by including memory access encryption, control flow flattening, and the insertion of opaque predicates.

TODO The malware evasion paper

4.2 Defensive Diversification: Speculative Side-channel protection

TODO Go around the last paper

4.2.1 Threat model

- Spectre timing cache attacks.
  - Rockiki paper on portable side channel in browsers.

4.2.2 Approach

- Use of wasm-mutate

4.2.3 Results

- Diminshing of BER

Contribution paper

The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review* `https://arxiv.org/pdf/2309.07638.pdf`.

TODO TBD discuss deoptimization

4.2.4 Partial input/output validation

When WASM-MUTATE generates a variant, it can be executed to check the input/output equivalence. If the variant has a _start function, both binaries, the original and the variant can be initialized. If the state of the memory, the globals and the stack is the same after executing the _start function, they are partially equivalent.

The _start function is easier to execute given its signature. It does not receive parameters. Therefore, it can be executed directly. Yet, since a WebAssembly program might contain more than one function that could be indistinctly called with and arbitrary number of parameters, we are not able to validate the whole program. Thus, we call the checking of the initialization of a Wasm variant, a partial validation.

4.2.5 Some other works to be cited along with the paper. Mostly in the Intro

Spectre and side-channel defenses

- paper 2021: Read this, since it is super related, `https://www.isecure-journal.com/article_136367_a3948a522c7c59c65b65fa87571fde7b.pdf` [**?**]
- A dataset of Wasm programs: [**?**]
- Papers 2020
- Papers 2019 - [**?**]

Selwasm: A code protection mechanism for webassembly

Babble

- https://arxiv.org/pdf/2212.04596.pdf

Principled Composition of Function Variants for Dynamic Software Diversity and Program Protection

- https://dl.acm.org/doi/10.1145/3551349.3559553

How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes

- https://ieeexplore.ieee.org/document/9975423

Code obfuscation against symbolic execution attacks

Code artificiality: A metric for the code stealth based on an n-gram model

Semantics-aware obfuscation scheme prediction for binary

Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly

Synthesizing Instruction Selection Rewrite Rules from RTL using SMT "We also synthesize integer rewrite rules from WebAssembly to RISC-V "

Wafl: Binary-only webassembly fuzzing with fast snapshots

4.3 Conclusions